EPISODE 1183

[INTRODUCTION]

[00:00:00] JM: TensorFlow Lite is an open source deep learning framework for on-device inference. TensorFlow lite was designed to improve the viability of machine learning applications on phones, sensors and other IoT devices. Pete Warden works on TensorFlow Lite at Google and he joins the show to talk about the world of machine learning applications and the necessary frameworks and devices that are necessary to build them.

[INTERVIEW]

[00:00:30] JM: Pete, welcome to the show.

[00:00:31] PW: Awesome. Thanks so much, Jeff.

[00:00:33] JM: Tell me, what are the resource constraints of machine learning frameworks?

[00:00:40] PW: So what's really interesting is that there are training constraints when you're actually trying to train your network, and that's usually to do with what GPUs or other server and cloud resources you have. And then there's the problem of, “Okay, I've got a trained model. How do I actually get it out into the world?” And what I've really been focused on for the last few years is how to get machine learning models off the cloud and onto things like phones, and even now microcontrollers. And there you not only don't have access to these sort of big beefy GPUs that you have in the cloud. You're often very limited on how much memory you have. If you're shipping an app, you don't want to increase the app size by 50 megabytes or something so that it doesn't fit in the app store. But you're also trying to run it on mobile phone CPUs. So you have a lot of compute limits as well. And those all get even more extreme once you start looking at something like an Arduino or some other kind of microcontroller. Then you're looking at you might only have 64k of memory to fit an entire

model into. So just to give you an idea of the order of magnitude of what you're looking at once you start deploying these models to the edge.

[00:02:18] JM: And if I train a really big model let's say in the cloud, does that mean that – Well, I have a lot of training examples that go into a model. Does that mean that the model is necessarily going to be big itself is the model size totally uncorrelated to the number of training examples that you give it?

[00:02:41] PW: It is pretty much uncorrelated with the number of training examples. Even comparatively small models almost always benefit from having more training data thrown at them. We have a model that is like 20 kilobytes in size. Has around like 20,000 parameters for doing voice recognition on microcontrollers, and we have about 100,000 utterances that we train it with. And honestly we could do with more training data to get better results. So, yeah, the amount of training data and the model size don't tend to correlate too much.

[00:03:30] JM: So what are the conditions where a model would be too big to fit onto an edge device?

[00:03:36] PW: So for most training situations, if you've got a model that's got hundreds of millions of parameters and so is hundreds of megabytes in size, that's perfectly fine on the cloud. So almost by default, models that you're training only with cloud deployment in mind are almost certainly going to be too large to fit onto mobile phones and microcontrollers.

[00:04:13] JM: And what are the typical techniques for slimming down a model?

[00:04:18] PW: So there are a whole world of research papers around – Just to name check some of the typical techniques, things like pruning, getting rid of weights. Things like quantization, which focuses on going from 32-bit float values down to much lower bit depth representations, typically 8-bit integers for the calculations. But honestly what we find to be most effective is actually starting off with an architecture that has been designed to be pretty

efficient in terms of its compute, in the amount of compute it does and also the number of parameters and the amount of memory it takes.

So what I usually recommend is looking around and trying to find something like, for example, in the image recognition space, MobileNet and the MobileNet family of models are almost as accurate as kind of the cutting edge imagenet models, but much, much smaller and involve much less compute. So what I usually recommend is don't worry about the model size when you start out. Just focus on the training data and focus on training any model architecture to get the results you want and get that working on the cloud. Because the hardest thing to get right really is the training data. So once you can train a big model, then start looking around at what people have done for more efficient, for more mobile and edge-focused architectures in the same area and then swap out the big architecture that you have been using to kind of prove that your training data is good enough and hopefully you should just be able to swap in the new architecture with the same training data and get something that's almost as accurate, but is much, much more efficient.

[00:06:27] JM: So the best practice is to change your model after training it and then retest it with the same training data and make sure you get the same results or close to the same results.

[00:06:39] PW: Exactly. And what I emphasize really is that the accuracy that you care about is very much a product of the app that you're actually deploying to. You have to consider not just the training kind of eval accuracy in the Python training loop. You have to actually consider how it works within the whole application.

And I think a good example of that is if you think about something like a translate app that's doing translation of images, of menus, of signs and things like that. If you have an offline model where you're sending an image to the server, that model might be super accurate. But the fact that you're having to wait several seconds to get the result back means that it's not very interactive.

So one of the interesting things that we found in situations like that that having a model that's actually running live on your phone to do that sort of translation can be perceived as much more accurate by users even though the model itself might be less accurate on strict eval classifications on the server side, because you're getting that interactive experience through running it on the phone at several frames per second.

So when you think about accuracy and when you're evaluating the model, really try and think about how you're actually running it as part of your whole app and as part of your whole system. Don't get too disheartened if you lose like one or two percent on like the top one accuracy when you move to a mobile friendly model, because you might actually gain in the perceived user experience because you're able to run that model much faster and have a much more interactive kind of friendly user experience.

[00:08:47] JM: So this approach to reducing the model size, is this the approach that TensorFlow Lite takes?

[00:08:54] PW: It is we actually offer what we call the model optimization toolkit to help users go from a training side model into something that's going to be deployed efficiently on the edge. So that especially focuses on actually doing quantization. So going down from the 32-bit float models that you typically use in the training environment down to the 8-bit integer quantized representation that works much better on edge devices.

[00:09:36] JM: So can you tell me a little bit more about the conversion process from a trained model to a reduced size model?

[00:09:46] PW: Yeah. So the first thing to do, like I said, is really try and make sure that you're starting off with a model architecture on the training side that is mobile friendly, that is edge friendly, because there're limits to what we can do kind of through an automated conversion process. If your model has hundreds of millions of parameters to start with, then it's not really possible to automatically shrink that down. So the first step is really trying to find an edge-friendly model. But then once you have that, there are variety of things that happen when

the TensorFlow Lite converter is run. One of them is actually fusing together some common ops so that they can be run more efficiently. Things like taking convolution and value and expressing them as kind of a single op. So there's some op fusion that happens there. There are also things that show up in the graph that are useful at training time but that aren't useful when you're actually just running pure inference jobs on the edge.

So an example of that is all of the back propagation. Updating the weights. Since the weights are frozen, you no longer need to have all the machinery to actually update them. And then there are things like batch normalization, where during training you're constantly updating the batch normalization values. But once you've trained the model, they're frozen. So you can actually take those operations that form the dynamic batch normalization and freeze them down into a much more efficient kind of constant based representation. And that's all before you start getting into things like quantization and some of the other things we look at to shrink stuff down.

So with the quantization, we actually have a couple of different techniques to do quantization because there are some requirements there where we need to know for quantization purposes what the ranges of the intermediate values in the graph are or the min and max values essentially of things like the activation layers. So we have a couple of different approaches to figure those out. One of them is called quantization aware training, which is where you actually modify the training process to produce more accurate quantized models at the end. And then we also have post-training quantization where you supply some representative data to an already trained model and then it figures out the quantization parameters from the representative dataset that you give it.

[00:12:55] JM: Could you maybe go through an example like some model example that you've worked with recently?

[00:13:03] PW: Yeah. So I will. After this podcast I'll actually post some links to some of the examples that we have up on GitHub and that you can actually find some collab tutorials for. We have a model that I mentioned, the sort of – I think it's like 20 or 30 kilobytes for the voice

recognition. And what we do there is, as I mentioned, we have these hundred thousand utterances of people saying different words, like several thousand different people saying a few dozen different words. We create a very small model in the training environment. So this model starts off in the only 20,000 parameters or something like that and it consists of a simple convolution layer followed by a fully connected layer. So it's a simpler convolution on your network as you can imagine really.

We then run training for a couple of hours. You can actually do that in the collab. And you end up with a model that's not record-breaking. It's not going to sort of blow your way in terms of its accuracy. But for the fact that it's only got 20,000 parameters, it can fit in like 20 or 30 kilobytes. It's still really useful and it's able to – With the default, it's actually able to recognize the words yes and the words no. So it's designed to be something that you can deploy on a microcontroller to control some simple project.

So once we've gone through the training process, you can see in the collab, we then go to the TensorFlow Lite converter and you invoke that from Python and you provide a series of examples of the sort of training data inputs that you would feed in during training, but you provide them as what we call a representative dataset so that the TF Lite converter is actually able to do a good job of figuring out the quantization ranges for the exporting when we export to this 8-bit representation of the original 32-bit floating point model.

And it also goes through and does things like gets rid of any batch norms, gets rid of things like drop out that aren't needed. Gets rid of all the back propagation and converts it into a different representation into the TensorFlow Lite serialization format. And then one of the things we actually do just to verify that the export has worked is we then load up the TensorFlow Lite interpreter in Python and run a bunch of examples of labeled utterances from the test set through this exported model to make sure it's working as expected.

And then once we're satisfied that we're actually getting the accuracy that we want, actually in this case, because we're trying to deploy on a microcontroller and microcontrollers generally don't have file systems. Instead of saving a file that contains the flat buffer, serialized version of

the TF Lite model, we actually take that data that would have been in the file and turn it into a constant C array that we can compile into our application. And that's really the sort of end-to-end export process.

[00:17:09] JM: And what parts of this process does the TensorFlow Lite framework make easier?

[00:17:15] PW: So one of the hardest parts over the last few years really has been the quantizing models. It's still a bit of an art rather than the science, because quantization is pretty hard to do automatically. So the model optimization team have done a great job really trying to push the state of the art on quantization and figure out how to actually make it a lot easier than it used to be to get 8-bit models out of your training process.

And there's also a lot of other optimizations that kind of happen under the hood like I mentioned around getting rid of back flop, batch normalization, getting rid of drop out and all of these other kind of details of the training process that you really don't want to have to worry about during inference. So the fact that the converter is able to take care of most of those for you is very helpful.

[00:18:31] JM: What's the workflow for somebody going from developing in TensorFlow to building their model into TensorFlow Lite?

[00:18:40] PW: So one of the challenges that we often find is that the people who develop models are often in a different team than the people who actually have to deploy the models in an application especially in an edge application. So in an ideal world, like I mentioned, there's actually a lot of iteration and feedback with the model creation process to actually find a model architecture that's going to be edge friendly.

So what we recommend is that there's kind of an integrated full stack sort of team that handles everything from data collection, to model training, to deployment on the device so that there's the opportunity to actually supply feedback and make improvements at all stages of the

process. Because, to be honest, like if you've got a model that's not edge friendly that you're just kind of handed and you don't get the opportunity to retrain, you're probably going to have a pretty bad deployment experience. You're going to have a model that's either too big or too slow. And it also may not have even been trained with data that reflects the sort of data that you'll be getting in your application.

So in terms of the workflow, what we really recommend is that you start doing exports of your model even before you've actually trained it just to check things like, "Okay, what's the size of this TF Lite model?" And you can do things like even with a model that hasn't got trained weights. You can still benchmark the latency and you can see how much memory it takes up in your phone or in your microcontroller application. And that can actually inform how you want to do the training.

And I definitely recommend trying to treat it not like the old sort of waterfall software engineering process where you have all these separate stages that happen in succession, but it's much more of an iterative process where you're constantly trying out new versions of the model in deployment and using what you've learned there to go back to the training process and to go back to the data collection process and labeling process even to figure out how to solve problems as they emerge. You're going to have a very bad experience if you're just kind of hoping to throw a model over the wall at the deployment team and expect it to work.

[00:21:47] JM: Are there any other frictions that you see between the machine learning, the different components of a machine learning team?

[00:21:55] PW: I mean whenever you have a group of engineers, there's always going to be some potential points of friction. I think that one of the things that's generally been tough has been the disconnect between the academic side of machine learning research where you expect to be working on a dataset that has been handed to you that is fixed. And the reality of working in applications and in industry where you actually have to spend honestly most of your time working on improving the dataset and you spend very little time on actually tweaking architectures.

So that can be a tricky transition for people who come from the academic side of machine learning research to kind of switch to thinking much more about, “Hey, how can we improve this dataset?” Versus, “Hey, how can I like tweak this architecture, this model architecture to provide better results?”

[00:23:07] JM: Does it ever happen that I train my model in TensorFlow? I modify it to TensorFlow Lite and it becomes too inaccurate?

[00:23:19] PW: So I mean that's definitely possible. It's not a very frequent problem that we see at least. What we do tend to see is sometimes people are using operations that are not supported in TensorFlow Lite in TensorFlow. So what tends to happen is that there're well over a thousand different operations in TensorFlow. And TensorFlow Lite doesn't support all of them. And researchers are constantly coming up with you know new variations and new kinds of operations. So it's much more likely that if you're using particularly obscure or sort of very, very new model, that there may be some lag in getting support for that in TensorFlow Lite. That seems to be more common than any concerns about accuracy.

[00:24:22] JM: Let's take a step back. Why is it important to deploy these models to edge devices? Why not just have big models in the cloud that we can just remotely access?

[00:24:35] PW: Yeah, that's actually a fantastic question. And when I started working on edge deployment back in 2014 or even earlier before I joined Google, that was a question that came up pretty often, was like, “Aren't these just cloud things? Shouldn't you just be running these on servers?” There are a whole bunch of reasons why there are advantages to running on the edge. One of them, like I mentioned with the translate example, is just the interactivity and the ability to have really low latency results provided to the user if you're running something on device. You can get down to the 20, 30 frames per second on something visual potentially, which you'd never be able to do even with 5G sort of network uh connections.

There's also just the economics and the physics of having connections to the cloud running all the time especially things like voice interfaces. A great example of using edge deployment to actually improve cloud-based experience is something like a voice interface that has a wake word, like a lot of the phone and other sort of voice assistants that we use around us. I won't say the OKG wake word in case it sets off a bunch of my devices and possibly people listening to this, but that is actually running locally on your device so that devices don't have to be sending like 99% silence over to the servers or 99% irrelevant speech over to the servers to deal with so that they could actually do the initial trigger running locally on device. And there are also a bunch of cases where especially in the microcontroller world where you have no guarantee that you actually have any kind of network connection at all.

There was a great project that's actually still continuing called Plant Village, which they used TensorFlow Lite to actually create an application that farmers in the developing world and especially in Africa could actually use to diagnose plant diseases for their cassava crops by pointing their phone at the leaves and getting a diagnosis of what was wrong with their plant and also getting advice on how they could actually improve that.

And in a lot of these places, there just is no connectivity. If you're out in the fields in the rural area of Africa, you may well not have a cell connection. And even if you do, it may be very expensive. You may want to not run down your battery. So having something that is actually able to run independent with the cloud is also a really important reason to have something that's deployed on the edge. And also it can be helpful to – Can be nice if you don't need to send data to the cloud to just keep it local. That's always helpful.

[00:28:25] JM: Would another on device example be the smart reply in Gmail?

[00:28:30] PW: Yeah. Yeah, that's actually a nice – It means that you have this very interactive experience. And, again, it doesn't rely on you having good connectivity. And one of the things I think about is that people do think of Google as being this very cloud-focused company. But if you think about some of the most impactful Google products, like Google Maps or Gmail, the thing that was really distinctive about them when they came out was that they had these

fantastic client-side experiences. With Google Maps, you could just click on the map and drag it around, and that was amazing at the time when it came out. And the same with Gmail, you suddenly went from having this very static HTML-based sort of view of your mail where you sort of clicked through to new pages to having this very JavaScript-driven interface that was very interactive.

And so I see on device ML as kind of a continuation of that tradition of bringing great user experience and bringing interactivity close to the user so that they can actually have a much better experience and have a very low latency, highly interactive, very intelligent interface running locally on their device that seamlessly integrates with all of the cloud infrastructure as well.

[00:30:05] JM: Once I have my model trained and I want to deploy the TensorFlow Lite model and then I want to update it over time. What is an update to an existing model on device look like?

[00:30:17] PW: So one of the nice things about TensorFlow Lite and sort of edge ML models in general is that you can swap out the model separately from the application code. So the model is just a file that defines the graph of computations that you want to run on device to execute the computation. So whether you're updating it with a new version or, for example, you might have a user select a particular language and download a different model depending on what language they actually want to work with. Replacing the model is just a matter of swapping out one data file for another data file. So that can be quite a nice experience when you're developing and updating your applications on the edge.

[00:31:17] JM: Are there any constraints around what kinds of devices I can deploy a TensorFlow Lite model to?

[00:31:24] PW: So we're trying to cover as many devices as possible. We have a lot of coverage of the smartphone world, so Android and iOS phones. We have a lot of people and a lot of applications and a lot of you know models being deployed. What we've been doing over

the last couple of years is then saying, “Okay, we've got these smartphones with gigabytes of memory and that can afford to have a couple of megabytes of models at least and hundreds of kilobytes of binary footprint for the code executable. Can we actually go down to much, much smaller devices and have sort of peel and stick sensors that maybe cost 50 cents? And so run on these sort of microcontrollers have – One of the goals has been, “Hey, can we have a voice interface that runs on a coin battery for a year on a piece of hardware that costs 50 cents?” Because we think that that would be really massive game changer in terms of how people can interact with all of the objects around them. So that's where TensorFlow Lite Micro has come in. And over the last couple of years we've actually pushed out the code and a whole bunch of examples on devices like the Arduino and the ESP32 that show how even if you've only got 64k of memory, say, you can still do useful things like a very simple voice interface on one of these microcontrollers.

So there really should be a flavor of TensorFlow Lite that you can use no matter what kind of edge device you're targeting, even something that's really pretty cheap and pretty resource-constrained. So I don't want to say there're no limits at all, but we've tried to make the limits as wide as possible on what you can actually deploy these TensorFlow Lite ML models on.

[00:33:45] JM: we talked a little bit about redeploying retrained models to devices after a successful deployment. But you can also just have learning at the edge. You can just improve models without doing a redeployment. Are there any frictions to doing that in a memory-constrained environment?

[00:34:08] PW: There definitely are, and I really haven't spent all that much time with the training side of the edge you know. Some of my colleagues have done some amazing work with federated learning, which is all about actually trying to learn on device and then actually send privacy preserving updates back to help improve the model for everyone else. It's not so much a resource constraint, though you do need to treat the weights as variables rather than constants if you're going to be training them. It just adds a whole another layer of complexity to the process. And it can be quite tough, because often you don't have the label data that you

have during training on the edge when you're actually trying to do some training updates. So just figuring out how to label the data so that you can use it in a traditional supervised learning process can be a big challenge in itself.

[00:35:19] JM: What are the unsolved problems in machine learning at the edge?

[00:35:25] PW: Oh! That is an extremely long list. I think a lot of the unsolved problems are around really trying to democratize this ML technology and make it widely available to all sorts of different app developers and people developing on embedded systems. At the moment, you still have to learn a fair bit about machine learning in order to create a model that you're going to deploy as part of your app or as part of your embedded system to solve a problem.

One of my dreams is that we can actually make it so that there are a lot more things that are along the lines of the AutoML product that you can get through Google Cloud where you don't actually need any ML knowledge to create a model. You can just supply some examples of the sort of training data that you care about and the automated process will actually produce a model for you and then you can just grab it and use it essentially as a black box component in the rest of your app or in the rest of your system. And if we're really going to have widespread usage of ML, I think we need to get to that point where for common problems around voice and image classification and accelerometers on the embedded side, we have some very, very developer friendly workflows that avoid you having to worry too much about what's happening under the hood.

[00:37:20] JM: And so what area of TensorFlow Lite are you specifically working on?

[00:37:25] PW: So I'm focused on shrinking TensorFlow Lite down to run on these embedded systems. So TensorFlow Lite Micro started off as what we call a research moonshot a couple of years ago, which is you know hey let's take a crazy idea that would be very high risk but would be really high value if we could pull it off and let's spend a year sort of researching it. And that turned into this version of TensorFlow Lite Micro, which will actually fit within less than 20 kilobytes of memory. So it's really this idea of taking this whole ML world and miniaturizing

it so that it will work on these very cheap, very low power, but very resource-constrained embedded systems so that we can end up in this world where hopefully we can have these sort of peel and stick sensors that we can have in agriculture. We can have them in the built environment in buildings around us. We can have them you know in wearables, in everything that we can actually interact with as people. We can have, as I mentioned, this idea of a cheap voice interface component that you can put into anything that we build. If we can replace switches and buttons with things like voice interfaces or gesture interfaces in all of these items, then I think that that's really going to change the whole way that we interact with the world around us. So that's what I'm really excited about as you might be able to tell.

[00:39:10] JM: Very interesting. And just to be clear for people. So what you're talking about is the runtime of you have these TensorFlow models that you deployed your edge devices, but there's a runtime that actually executes those models.

[00:39:25] PW: Exactly. And it's an interpreter that takes the model files and calls the calculations needed to execute the model based on sort of the trained weights. And it also includes optimized implementations of those calculations for all of the different platforms that we care about like the ARM Cortex-M or the ESP32 or the Arduino, all of these different platforms that people are using on the embedded side.

[00:39:59] JM: Cool. Well, anything else you'd like to add about TensorFlow Lite or the ecosystem in general?

[00:40:06] PW: So there's actually a lot of tutorials that you can actually find that will help you especially on the Arduino side if you're an Arduino fan. You can find the TensorFlow Lite Micro Arduino Library as one of the official Arduino libraries that you can download. And we actually also have an edX course that's being run by Harvard in collaboration with them that will take you through a lot of the edge deployment stuff that I've been talking about. So I'll post some links to some of that material up on Software Daily after the podcast. And yeah, I'm really looking forward to seeing what people end up building with all of this stuff.

[00:40:53] JM: Great. Sounds like a great place to close off. Well, thanks for coming the show. It's been a real pleasure.

[00:40:57] PW: Awesome. Thanks so much, Jeff.

[END]